Language- and Machine-Independent Global Optimization on Intermediate Code

Bal, H.E.; Tanenbaum, A.S.

published in
Computer Languages
1986

DOI (link to publisher)
10.1016/0096-0551(86)90004-4

document version
Publisher's PDF, also known as Version of record

Comput. Lang. Vol. 11, No. 2, pp. 105–121, 1986
Printed in Great Britain

0096-0551/86 $3.00 + 0.00
Pergamon Journals Ltd

LANGUAGE- AND MACHINE-INDEPENDENT GLOBAL OPTIMIZATION ON INTERMEDIATE CODE

HENRI E. BAL and ANDREW S. TANENBAUM
Department of Mathematics and Computer Science, Vrije Universiteit, Postbus 7161,
1007 MC Amsterdam, The Netherlands

(*Received* 22 *January* 1986)

Abstract—Many retargetable production compilers use some form on intermediate code for applying global optimizations. The compilers of the Amsterdam Compiler Kit use a low-level, language- and machine-independent, intermediate code called EM. The choice of intermediate code has impact on both the effectiveness of a global optimizer and on the ease of achieving a machine-independent implementation. These effects have been studied by implementing several global optimization techniques for EM. Comparisons are made with optimizing codes of a higher-level (e.g. trees) or of a lower level (e.g. assembly code). It is also shown what kind of knowledge about the source language and target machine a global optimizer should have, to be effective.

Global optimizer Intermediate code Retargetable compiler

1. INTRODUCTION

The subject of code optimization has gone through a long process of development during the last two decades. Traditional research has focussed on finding new optimization techniques for some combination of source language and target machine and on improving existing ones [1]. More recently there has been growing interest in compilers that can easily be adapted to new target machines or even to new programming languages. Because of this shift of interest, it has become more important to find optimizations that can be applied to a large set of machines, using some form of parameterization. Such optimizations are usually called "machine-independent" or "machine-relative" [2].

Our research aims to test the feasibility of the idea of using a language- and machine-independent intermediate code for applying existing optimizations in a language- and machine-independent way. It was shown in an earlier paper [3] that many peephole optimizations can be done in such a way. In the current paper we will consider global optimizations, that analyze and change larger portions of program text than just straight-line code. In particular, global optimizations may consider and take into account the flow of control not only within procedures, but between procedures as well.

2. A COMPARISON OF SOME RETARGETABLE OPTIMIZING COMPILERS

Our research takes place in the framework of the Amsterdam Compiler Kit (ACK) project. We will first compare the approach of ACK with those of other retargetable optimizing compilers. These compilers were selected, because they are all different in design and all represent the current state-of-the-art in compiler building. Other compilers that aim at achieving language- or machine-independency are reported in Refs [4–6].

Most notably, these compilers differ in their choice of intermediate code use in the translation process from source program to object code. An intermediate code is called a high level code if it still contains the syntactic structure of the source program. Such representations often have the form of a tree [7, p. 86] or a graph [8]. In a low level intermediate code this structure is 'flattened' into a linear sequence of operations (e.g., a loop is expressed as a set of tests and branches). Triples and quadruples are examples of such code. Although trees and linear representations are not fundamentally different, for some optimizations one representation may be more convenient than another. Assembly code can be regarded as a low level code that is specific to a single machine.

2.1. *The Portable C Compiler*

The approach used by the Portable C Compiler (PCC) [9] is to write a well-structured compiler that is largely machine-independent. PCC consists of two passes. The intermediate code of the passes is a mixture of assembly code (for control structures) and trees (for expressions). The first pass, which is mostly machine-independent, performs some simple special-case optimizations on expression trees, such as constant folding, local strength reduction and removing useless additions (by 0) and multiplications (by 1). The second pass generates assembly code from expression trees. It is template driven and uses Sethi–Ullman numbers [10] to obtain efficient code. A separate program can be used to do some peephole optimizations on the assembly code.

To retarget PCC, the machine-dependent parts have to be rewritten and new templates for the code generator must be made. This accounts for about 25% of the compiler [9]. The peephole optimizer has to be rewritten for every new machine.

2.2. *The Retargetable Peephole Optimizer Project*

The design philosophy of Fraser and Davidson [11, 12] is to use a very simple code generator producing correct, but inefficient assembly code, followed by a powerful peephole optimizer (called PO) that generates production quality code. As the code generators are simple and easy to retarget and the optimizer is machine-independent, the process of retargeting the entire compiler is straight foward. A compiler for the "Y" programming language has been built using this approach.

The optimizer uses a formal description of the target machine's instruction set. The semantics of every assembler instruction is expressed as a sequence of register transfers. The optimizer tries to combine sequences of assembler instructions into single instructions, by symbolically manipulating the corresponding register transfers. The optimizer is not pattern driven, but performs an exhaustive search for sequences of a limited length (pairs and triples).

The process can be speeded up by using this sytem to find patterns that occur frequently in user programs, and replacing the exhaustive search by a much faster pattern matcher [13].

2.3. *The Production-Quality Compiler-Compiler Project*

The PQCC Project [14] is a large and ambitious project that aims at automating the construction of compilers, especially the machine-dependent parts. PQCC compilers use an abstract syntax tree (called TCOL) as intermediate representation and do massive global optimization on this tree. Some optimization also takes place after code generation.

Machine dependencies are isolated in a set of tables that the compiler-compiler system generates from machine descriptions. Retargeting involves writing new machine descriptions.

2.4. *The Amsterdam Compiler Kit Project*

The Amsterdam Compiler Kit (ACK) is based on the UNCOL idea [15] of using a low level language- and machine-independent intermediate code (an UNCOL). Machine independency of the optimizations is obtained by applying them to the intermediate code. The ACK code generators are table driven: virtually all machine dependencies are stored in an internal driving table. Retargeting an ACK compiler consists mainly of writing a new machine description table that is automatically converted into the internal driving table of the code generator.

2.5. *Comparison*

All four systems use fundamentally different approaches to achieve high quality code and to ease retargeting.

PCC does little optimization, but aims at producing good local code. The remaining compilers emphasize optimizations on respectively assembly code (PO), tree intermediate code (PQCC), and low level intermediate code (ACK).

PCC eases the task of retargeting by isolating machine dependent code and by using templates for code generation. The peephole optimizer is not machine-independent. The PO system attempts to make code generation trivial. It provides a machine-independent optimizer using a machine description. PQCC uses phase generators to build all compiler phases from machine description tables. ACK uses a machine-independent intermediate code for optimization and provides a tool for making code generators.

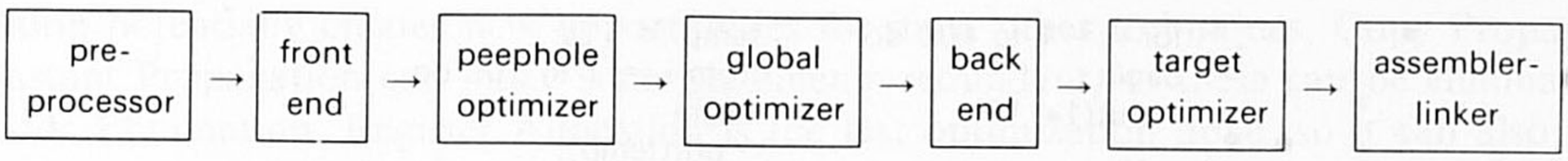

Fig. 1. ACK compilation process.

3. OVERVIEW OF THE ACK COMPILATION PROCESS

The infrastructure of an ACK compiler consists of a front end that translates the source program into a low level intermediate code called EM and a back end that translates EM into the assembly code of the target machine. Additionally, a compilation may involve a preprocessor (macro processor), an assembler/linker and several optimizers. The entire compilation process is depicted in Fig. 1.

The preprocessor and peephole optimizer can be used for any language and machine. A front end needs to be written for every new language. All remaining phases use tables containing machine-dependent information and can be adapted to a new machine with little effort. The compiler phases are highly independent. All information that is carried along is expressed in the EM code.

Optimizations take place at several stages of this process. The front end usually does little optimization. The "C" front end must be able to compute static expressions in some contexts, so it evaluates all these expressions (i.e. constant folding). The peephole optimizer performs many optimizations that can be expressed as pattern replacements. Local special case analysis can be added to the driving tables of the back ends. The target optimizer only does simple (probably machine specific) optimizations that cannot be easily done in earlier phases.

EM is a low level intermediate code, well suited for algebraic languages (so not for Cobol or Snobol) and byte addressable machines (not a PDP-8 or Cyber 170). This covers a large class of languages and machines. At present, we have front ends for Pascal, "C", and Basic and back ends for the PDP-11, VAX, 68000, NS16032, Z8000, Z80, 8086, 8080, and 6502. Several universities are using EM for implementing other languages, such as Algol 68, DAS (Delftse Ada* Subset), Plain, and Y. We ourselves are currently working on a Modula-2 front end.

EM has the architecture of a stack machine, with a very simple storage model, lacking general pupose registers. Local variables are addressed via an offset in a stack frame. (Storage binding is done by the front ends). Local variables of a lexically enclosing procedure are accessed via a static chain. EM has an extensive set of instructions for manipulating local or global variables, doing arithmetic and altering the flow of control. It also has special case instructions such as an INCREMENT VARIABLE. In this sense EM is a union machine rather than an intersection machine [11].

The EM assembly code also supports *pseudo instructions*. For example, a front end or optimizer can generate a *register message* pseudo, indicating that the code generator should put a certain variable in a register. So the lack of general purpose registers in EM by no means prevents ACK compilers from efficiently using the target machine's registers.

This description of EM is sufficient for reading this paper. However, interested readers are referred to Refs [16–18] for more details of EM and ACK.

4. THE EM GLOBAL OPTIMIZER

Global optimizations in ACK are only done on the intermediate code (EM). The motive for this is the tendency of global optimizations to be much harder to implement than local or peephole optimizations, so the implementation should be done only once and must be useful to all languages and machines supported by ACK. As our interests lie in making compilers easy to retarget, the optimizer should use as little information about the target machine as possible, although some information will be shown to be indispensable.

The optimizer will perform better if the entire program (in EM format) is presented to it at once,

---

*Ada is a registered trademark of the U.S. Government (Ada Joint Program Office).

```
for i := 1 to 100 do
begin
  put(1*118);
  ...
end;
```

(a)

```
temp := 118;
from i := 1 to 100 do
begin
  put(temp);
  temp := temp + 118;
  ...
end;
```

(b)

Fig. 2. An example of Strength Reduction (on source code level): (a) unoptimized source; (b) optimized source.

but this is certainly not a requirement. Neither does it not require the user to supply an execution profiles. The user may ask to optimize for execution time or for code size.

The optimizations performed by the global optimizer will be described briefly below.

—Inline Substitution [19] reduces the execution time overhead involved in procedure calls. Very small procedures and procedures that are called only once can always be expanded in line, possibly even decreasing the size of the program. It is also possible to sacrifice space in favour of execution time, yielding a larger yet faster program.

—Common Subexpression Elimination [20] is a classical optimization that reduces execution time by eliminating multiple computations of the same expression. It saves the result of the first occurrence and uses this value for every recurrence.

—Strength Reduction in loops [21] replaces the repeated use of an expensive operation (e.g. a multiplication or an array reference) by a cheaper one (an addition and an indirect pointer reference). (See Fig. 2.)

—Stack Pollution reduces execution time and instruction space at the cost of some stack space. In EM, parameters of a procedure call are passed via the stack and are removed (popped) by the calling procedure. As procedure calls tended to occur frequently, this stack cleanup accounts for a significant part of object code size and execution time. In many occasions this cleanup can be deferred, so several cleanups can be combined into one. (See Fig. 3.)

—Cross Jumping (also called tail merging) [22] eliminates code that appears on two control flow paths that come together at a certain point (see Fig. 5 in Section 6).

—Copy Propagation [23] tries to make statements of the form A: = B redundant by replacing subsequent references to A by references to B, provided that A and B are not changed.

—Constant Propagation [23] is similar to Copy Propagation. After a statement A: = constant, it replaces references to A by the constant, as long as A is not assigned a new value.

—Dead Code Elimination [23] removes assignment statements that are useless because the assigned variable is not subsequently used, probably as a result of Copy Propagation and Constant Propagation.

—Register Allocation [24] tries to use the target machine's general purpose registers as efficiently as possible. Without the intervention of the global optimizer, only local variables are put in a register, for the duration of an entire procedure. This scheme can be improved significantly by putting other entities (such as constants and addresses of procedures or variables) in registers and by packing several entities in one register, for example putting different variables in the same register during the first and second halves of a procedure.

The optimizations are performed (by default) in the order they were presented above. The order is important, as some optimizations create or destroy opportunities for other optimizations. Inline

```
p(2,3);
q(7);
```

(a)

```
PUSH 2
PUSH 3
CALL p
POP  4 bytes
PUSH 7
CALL q
POP  2 bytes
```

(b)

```
PUSH 2
PUSH 3
CALL p
PUSH 7
CALL q
POP  6 bytes
```

(c)

Fig. 3. An example of stack pollution with a 16 bit machine: (a) source program; (b) unoptimized (pseudo) EM; (c) optimized EM.

Substitution potentially creates new opportunities for most other techniques. Copy Propagation and Constant Propagation can make some statements redundant, so these can be eliminated by Dead Code Elimination. Register Allocation is the last optimization done, so it can also assign registers to temporary variables created by other optimizations.

Some optimizations that could have been done at the EM code were left out because of time limitations of the research. These include several loop transformations (loop unrolling, loop fusion and loop splitting [25]), code motion out of loops, and code hoisting [23]. We have also excluded very machine-specific optimizations (such as using exotic instructions [26]) as they cannot be expressed in EM. Evaluation order determination [14] is not done, as the order of computations is already determined in the EM code, and the code contains no information about possible reorderings.

5. CONTROL FLOW ANALYSIS ON INTERMEDIATE CODE

The most important piece of information maintained by a global optimizer is a description of the control flow of the input program. Control flow information specifies the order in which the parts of the program may be executed dynamically. In a tree representation of the source program, most of the control flow is implicit in the structure of the tree, as the tree contains high level control statements, like for-and if statements. One notable exception, however, is the explicit goto statement.

A low level representation contains only explicit (possibly conditional) jumps. In order for the control flow to be determinable, the set of possible targets of every jump must be deducible from the program text at compile time. EM has conditional, unconditional, subroutine, case, and non-local jumps. A case jump branches via an index table; a non-local goto may jump out of the current procedure. The latter two provide a handle for indirect jumps, (i.e. a jump to a runtime constructed address). The global optimizer inhibits this possibility by requiring the case index table and the target address of the non-local goto to be unmodifiable. Hence an unconditional and non-local jump branch to a single target, a conditional jump either branches to a specific label or continues at the textually next instruction and a case jump may branch to labels contained in the index table.

So, in principle, the control flow is determinable from the EM code. Yet there are still several ways to "deceive" the flow of control. In some languages the user is allowed to specify what actions should be undertaken when an exception (interrupt or error) occurs. If the language allows the interrupted computations to be resumed after the exception has been handled, this effectively means that the exception handling code can be inserted dynamically at almost any point of the program. Hence, even very simple optimization, like:

```
A:=10              A:=10        (1)
B:=X[I]     →      B:=X[I]      (2)
C:=A               C:=10        (3)
```

cannot be done, as an exception may occur at line (2), and the exception handling code may assign a new value to A. This danger has been clearly understood in the design of Ada, but not in that of PL/I [27].

Another way to cheat the control flow is by using assembly routines. An example is the use of the notorious "setjmp" and "longjmp" routines, that are part of "C" runtime library [28]. They can be used (relatively cleanly) to jump out of a procedure, or for any other number of filthy puposes.

The global optimizer does not defend itself against such attacks. Only a very thorough approach (such as Spillman [29] uses for PL/I) would be efficacious, but it would also make the optimizer totally ineffective. Programs that use such extreme features simply should not be optimized by a global optimizer.

From the control flow information it is simple to detect loops in the programs. As programs tend to spend most of their time in loops, this knowledge is vital to time optimizations. In a tree representation, the loops are already available, although this information may be rather deceiving in the presence of jumps. In Fig. 4, the only code that is really part of the loop is the test for the

```
while true do
being
        if cond then
        begin
                {lots of simple
                 statements}
                goto exit; {exit loop}
        end;
        S; {one statement}
end;
exit:
```

Fig. 4. A deceiving control structure.

condition and the statement S, although the remainder of the code is textually part of the loop too. That is, only the condition and S can be executed more than once.

6. OPTIMIZATION ON INTERMEDIATE CODE

The main benefit of using a language- and machine-independent intermediate code is the fact that it eases the implementation of a language- and machine-independent optimizer. A machine-independent optimizer that operates on assembly code or object code, such as the PO optimizer discussed in Section 2.2, must be able to deal with various non-orthogonal instruction sets of the target processors. Also, most machines have a complex storage model, probably including registers with curious properties (such as using only odd-numbered registeres for multiplication). The presence of condition codes can be a nuisance too. On the other hand, higher level intermediate codes are not directly related to any particular machine, so these problems can be avoided. EM has an orthogonal instruction set. It does not have general registers or condition codes at all, making optimization much simpler.

As a drawback of the intermediate code approach, it is often difficult to achieve the same effectiveness as can be obtained by optimizing assembly or object code. This effectiveness depends on:

(1) the level of detail of the intermediate code;
(2) the information about the source language and source program that is available in the intermediate code;
(3) the available information about the target machine.

```
SOURCE:                 INTERM. CODE:                ASSEMBLY CODE:
if c then                       TEST c                       cmp -2(bp),0
  p(10)                         JMP_ON_FALSE LAB1            je labl
else                            PUSH 10                      mov ax,10
  p(20)                         CALL p                       push ax
end if                          POP word                     call p
                                JMP LAB2                     pop si
                          LAB1: PUSH 20                      jmp lab2
                                CALL p                 lab1: mov ax,20
                                POP word                     push ax
                          LAB2:                              call p
                                                             pop si
                                                       lab2:
                                  |                            |
                                  v                            v
                                TEST c                       cmp -2(bp),0
                                JMP_ON_FALSE LAB1            je labl
                                PUSH 10                      mov ax,10
                                JMP LAB2                     jmp lab2
                          LAB1: PUSH 20                lab1: mov ax,20
                          LAB2: CALL p                 lab2: push ax
                                POP word                     call p
                                                             pop si
        (a)                     (b)                          (c)
```

Fig. 5. A Cross Jumping example: (a) source program; (b) CJ performed on low level code; (c) CJ performed on 8086 assembly code.

```
main( ) {                    main( ) {
    int a,b,*p;                  int a,b,*p;
    ....                         ....
    a = 10;                      a = 10;
    *p = 20;                     *p = 20;
    a = a;                       b = 10;
    ....                         ....
}                            }
      (a)                          (b)
```

Fig. 6. A dubious Constant Propagation optimization: (a) unoptimized "C" source; (b) optimized "C" source.

We will first demonstrate the importance of these factors by giving clear examples. The statements made here will play a key role in the discussion of the implementation of the global optimizer later on.

6.1. *Level of detail of the intermediate code*

At a low level, the actions defined by the source program are made more explicit, so they are expressed in smaller atomic units. Consequently, at a low level opportunities will arise that could not be easily anticipated at a higher level. As an example, consider the Cross Jumping optimization technique, applied to the program of Fig. 5(a).

This optimization tries to identify a common piece of code in the tails of the true-part and the false-part of the if statement. At the source level (and at the parse tree level) there is not such common piece of code [Fig. 5(a)]. At the low intermediate code level (Fig. 5(b)], both subroutine calls are translated into three actions; push the parameters, call the subroutine and pop the parameters*. The latter two actions form a common tail. In the assembly code for the 8086 [Fig. 5(c)] the "push parameter" action is translated into two assembly instructions, as the 8086 lacks a "push constant" instruction. The second instruction can be included in the common tail. So Cross Jumping is more effective when it is applied to assembly code.

EM has the same level of detail as the code of Fig. 5(b).

6.2. *Source language information*

Often, an optimization can only be done effectively if some specific property of the source program is known. This information may be obtained at an early stage of compilation and can subsequently be propagated in the intermediate representations.

The most important aspect of the source program concerns the types of the program variables. Especially information about the usage of pointers is of vital importance. Optimizers are often impaired in the presence of indirect references (through pointers). The "C" programs in Figs 6(a) and 6(b) are only equivalent if the indirect assignment through "p" cannot change "a".

Type information can be used to decrease the destructive influence of pointers. For example, in a strongly typed language like Pascal, a pointer variable can only point to objects of a certain type, so an indirect assignment through such a variable can never affect objects of a different type. Many optimizations will benefit from this information. Register Allocation will in general only be possible, if one can make sure that some variables are never accessed indirectly, as the registers of most target machines cannot be accessed indirectly. Unfortunately, type information is often omitted in low level intermediate codes.

The EM code does not contain a complete description of the types in the source program. However, it does have a mechanism to decrease the negative effects of indirect pointer references. The EM code contains so called 'register messages', indicating that certain local variables are never referenced indirectly. These messages are generated by the front ends. For example, the "C" front end generates such a message whenever it finds a local variable to which the "address-of" operator ("&") is never applied. In our Pascal implementation, the address of a local variable is only used when the variable is passed by reference in a procedure call or when the variable is accessed from a nested procedure. The optimization of Fig. 6 would only be done if a register message for "a" was found.

---

*We assume here that parameters are passed via the stack and are removed by the calling procedure.

A register message also contains the size of variable, some indication of its type (pointer, floating point or integer) and a count of its occurrences. When a user specifies that the global optimizer is not to be invoked (to speed up compilation during debugging), the counts are used to assign registers to often used variables.

6.3. *Target machine information*

Knowledge about the target machine is needed:

(1) to decide which feasible optimizations are really desirable;
(2) to choose the most effective optimizations, if some selection has to be done.

Desirability analysis is needed to check if a certain transformation is advantageous for a given target machine. Optimizations are usually called "machine-independent" if their feasibility can be proven independent of any target computer. Yet, with enough morbidity, even the most "machine-independent" optimizations can often be shown to be disadvantageous for some target machine in some special case. As a clear example, consider the (local) optimization of Constant Folding, which is the computation of static expressions:

$$125 + 5 \rightarrow 130$$

Now, consider two ways to this computation on the 68000 (assuming 32-bit integer arithmetic):

```
move.1  # 125,d0        move.1  # 130,d0
add.1   # 5,d0
```

The 68000 has a special short (2 byte) instruction to load a constant between -128 and $+127$ into a register and another one to add a constant in the range 1–8 to a register. Hence, the code on the left hand side takes 4 bytes. The single instruction that moves 130 into d0 takes 6 bytes. So a programmer who wishes to minimize the size of his program will not be too impressed by this 'optimization'. In fact, the original source language statement "i: = 130" can best be optimized by splitting it up into two machine instructions, as shown above.

A second purpose of using machine-dependent information is in selecting the most effective optimizations when there is some limit on the optimizations that can be done. The usual sequence of events for some optimization technique is first to find an opportunity for optimization (such as a common subexpression or a dead statement), then to check if the transformation is really desirable, and finally to apply the optimization. For techniques like Inline Substitution and Register Allocation, the opportunities are numerous and easy to find. Here, the main problem is caused by a constraint imposed on the amount of optimization that may be done. The goal for these optimizations is to maximize their profits while obeying some restrictions.

The EM global optimizer uses a *machine* descriptor file to contain machine-dependent facts. Each descriptor file contains some general properties of the target machine (word size and pointer size) and information needed by some specific optimization techniques. The layout of the descriptor file is shown in Fig. 7. The entries will be explained in the next section. The descriptor file is kept as small as possible, so little effort is needed to add new machines.

7. IMPLEMENTATION OF THE EM GLOBAL OPTIMIZER

The global optimizer for EM specified in section 4 has been implemented in the "C" [30] programming language. We will first outline the overall design of the optimizer and then discuss all optimization techniques implemented by the optimizer. More details of the implementation can be found in [31].

7.1 *Overall design of the optimizer*

The optimizer consists of several phases, each doing one or more related optimizations. All techniques are implemented independently of each other. This means, for example, that there are distinct algorithms for Register Allocation and for Common Subexpression Elimination, rather than one algorithm doing both simultaneously [32]. Temporary variables created by Common Subexpression Elimination are indistinguishable from user-defined variables and compete with

```
GENERAL INFORMATION:
    word size
    pointer size
INFORMATION FOR REGISTER ALLOCATION:
    # general registers
    # address registers
    # floating point registers
    costs saved by putting a variable/constant/address in a register
    costs of initializing a register with a variable/constant/address
    costs of saving and restoring N registers
INFORMATION FOR COPY PROPAGATION:
    access costs for local variables
    access costs for global variables
INFORMATION FOR COMMON SUBEXPRESSION ELIMINATION:
    cheap EM instructions
    special cases that should not be optimized
```

Fig. 7. Layout of the machine descriptor file.

them for registers on equal terms. The order of the phases is rather flexible. The default order was given in Section 4.

The single most important data structure of the optimizer is the control flow graph, which represents the control flow information. The modes of this graph consist of basic blocks, which are linear pieces of single entry, single exit EM code. Each node has a set of successors, the nodes that can follow it immediately during execution. (We also maintain the inverse, predecessor, relation.) The immediate dominator of each node is computed by the algorithm of [33] and is used to detect loops in the graph [23].

Another important data structure is the call graph, specifying for every procedure the procedures it calls directly. Some interprocedural analysis is maintained, such as the set of all global variables affected by a call to a procedure P. This information is computed using a transitive closure over the call graph.

Reaching-definitions and live/dead information [23] are obtained via data flow analysis on the control flow graph [20]. The former specifies for every basic block the definitions (assignments) that reach the beginning of that block. Live/dead information specifies for any point of text the variables that are live (i.e. have a value that may subsequently be used).

A major problem in the design of a global optimizer is to keep all this information up to date after a program transformation. The problem can be avoided if one is willing to recompute from scratch any information that may have become obsolete. In general, the EM global optimizer tries to maintain the control flow graph, the call graph and the interprocedural information. An exception to this rule is Inline Substitution, that drastically alters the structure of the program. So, control flow analysis is simply repeated after Inline Substitution. Reaching-definitions information is only used by Copy Propagation and Constant Propagation, which are implemented in one phase. Live/dead information is generated by Dead Code Elimination and is used by Register Allocation, so these two phases are run consecutively.

7.2 *Common Subexpression Elimination*

Common Subexpression Elimination tries to eliminate multiple computations of EM expressions that yield the same result. It places the result of one such computation in a temporary variable, and replaces the other computations by a reference to this temporary variable. The primary goal of this technique is to decrease the execution time of the program, but in general it will save space too. The implementation uses the well-known *value number method* [20].

The most interesting aspect of the implementation is the desirability analysis. The elimination of a common subexpression may be undesirable if an extra instruction is needed to save the result of the expression, while the expression itself is cheap already. Even worse, the expression may be computed almost for free using a special addressing mode. A well-known source of common subexpressions is in subscript calculation for array references, such as

```
X:=A[I]+B[I]
```

If the element size of the arrays is 4 bytes, there is an implicit common subexpression "4*I". In an intermediate code the expression may be made explicit and can be eliminated. On a machine that has a scaled indexed addressing mode, like the VAX, the original statement can be translated much more efficiently, however, as this addressing mode automatically does the multiplication. Especially on a three address machine like the VAX, small expressions are often better retained. Ideally, desirability analysis should take into account the costs of saving the result after the first computation, the gain (if any) of using the result instead of recomputing it, the number of recurrences of the expression, and the context of the expression.

The costs of an expression depend on the target machine, the code generator and the context of the expression. Estimating these costs from the EM code of the expression is hard to do, as it would require detailed knowledge of the target machine instruction set with all its inorthogonalities, and of all fancy tricks done by the code generator. Feeding the EM code to the code generator and then analyzing the assembly code is not appealing either, and it does not even solve all problems, as the expression may not be in the right context. This context consists of any surrounding piece of code that may have impact on the cost of the expression. If one thinks of a machine like the 68000, where locations below 64 K are cheaper to access than other ones, one may not be surprised that the only correct context will ultimately be the entire program.

We have developed a simple, reasonably effective solution. Some operators should never be eliminated, as they are implemented cheaply on the target machine. Such operators (e.g. arithmetic conversion operators) can be specified in the machine description file. There are only a few cases where analysis of the context of the operator is needed. For example, on the VAX, the expression "4*I" should not be eliminated if it is an array subscript. The optimizer recognizes special cases that are disadvantageous on at least one target machine. Whenever it wants to eliminate an expression, it checks if the expression is such a special case, and if so it consults the machine description file to see if the elimination should be done. At present, there are only three such special cases in the table.

7.3 *Cross Jumping*

The Cross Jumping optimization scans the control flow graph for pairs of basic blocks (B, B2) that have one and the same successor block. In the source program, B1 and B2 may correspond to the true- and false-part of an if-statement or to two cases of a case-statement. If B1 and B2 end in the same sequence of instructions, one such sequence can be eliminated, as shown in Fig. 5(b).

Cross Jumping only removes code, but it does add instruction labels. If such a label is put somewhere in the middle of an expression, the computation of this expression will hence be split over two basic blocks. As the ACK code generators analyze one basic block at a time, certain other single-block optimizations will be inhibited, which may result in poorer code. The optimizer therefore avoids doing Cross Jumping transformations that split expressions into two. Note, however, that this analysis is not machine dependent. It only depends on the strategy of the code generators, which is the same for all target machines.

7.4 *Strength Reduction*

Strength Reduction looks for variables whose values form an arithmetic progression at the beginning of the loop. These variables are called induction variables. Intuitively an induction variable corresponds to a loop variable in a high-order programming language. Expressions that use a linear function of an induction variable as operand of a multiplication or as index in an array subscript can be made faster (see Fig. 2). Some programming languages have other operators that could be reduced in strength too, such as the exponentiation operator. Unfortunately, EM does not contain operators that are only present in a few programming languages, so these optimizations cannot be done on the EM code.

The transformed program will in general execute faster. If the transformed loop is executed zero times, the execution time of the program will be increased a little, as initializing code is added outside the loop. It is not generally possible to predict whether a loop will be executed at least once. Strength Reduction uses an 'optimistic guess' and always does the transformation (provided that the sematics of the program are not altered).

7.5 *Copy Propagation*

Copy Propagation analyzes every usage as operand of any (scalar) variable A. For every occurrence U, it checks the following conditions:

(1) the value of A at U can only have been assigned to it at the single statement D, of the form "A: = B" (called a *copy*);
(2) the value of B is not changed in between D and U.

If both conditions are satisfied, A can be replaced by B.

If Copy Propagation happens to replace all subsequent uses of A after a copy "A: = B", the copy statement becomes useless (as A is not longer used) and can be eliminated by the Dead Code Elimination optimization. Copy Propagation is only useful if A is more expensive to access than B, or if the copy statement can be eliminated. The access costs of A and B depend on whether they are local, global, or register variables. Even within these three classes there may be differences. On the VAX, both the object code size and the time to access a local variable depend on the offset of the variable in the stack frame of its procedure, and on the alignment of the variable.

Copy Propagation does not know if A and B will be stored in registers or if the copy "A: = B" can be eliminated, as these decisions are made by later phases of the optimizer. Therefore, heuristic rules are used to decide if the transformation is desirable. The access costs of A and B are estimated, assuming neither of them is put in a register. Machine-dependent information stored in the machine description file is used here. If B is cheaper to access than A, the optimization is always done. If B is more expensive, the transformation is omitted. If A and B are equally expensive, Copy Propagation applies the heuristic rule to replace infrequently used variables by frequently used ones. This rule increases the chances of the copy to become useless. Furthermore, B is more likely to be assigned a register than A, since the register allocation algorithm uses the static frequencies as input.

7.6 *Constant Propagation*

Constant Propagation is similar to Copy Propagation. For every usage of a variable A it checks if the value of A can only be assigned to it by a single statement of the form "A: = constant". If so, A can be replaced by the constant.

In principle, a constant can be more expensive to use than a variable. Most notably, the variable may be a register variable. On the other hand, having a constant as an operand may create new opportunities for other optimizations, so Constant Propagation is optimistic and always does the transformation.

7.7 *Dead Code Elimination*

Dead Code Elimination is a simple optimization that can be used to remove the inefficiencies introduced by other techniques. It looks for assignment statements with a left hand side variable that is dead after the assignment (i.e. its value is not subsequently used). If the right hand side has no side effects, the entire statement can be deleted. Dead Code Elimination only removes code, so it is unlikely that it will ever be disadvantageous. Hence it is fully machine-independent.

7.8 *Stack Pollution*

Stack Pollution is applied to one basic block at a time. It repeatedly tries to combine two successive stack cleanups into one (see Fig. 3). The new cleanup can be combined with the next one, and so on. In EM, a stack cleanup is easily recognized, as EM has a special instruction for it, called ASP (Adjust Stack Pointer).

As EM is a stack machine, the stack is not only used for passing parameters, but also for doing arithmetic and for storing intermediate results of expressions. So it is only safe to combine two cleanups if the following conditions are satisfied:

(1) No item pushed onto the stack before the first cleanup is popped between the two cleanups;
(2) The second cleanup must pop as many bytes as were pushed by the first one.

Condition 1. asserts that the first cleanup may be deferred. *Condition 2.* asserts that the combined cleanup pops exactly as many bytes as two separate cleanups would have done.

As Stack Pollution only deletes code, the resulting program will always be smaller and faster. The penalty is a slight increase in stack space. If this is inadmissable, the entire optimization should simply be skipped. Hence, Stack Pollution does not need any language- or machine-dependent information.

7.9 *Inline Substitution*

Inline Substitution replaces some procedure calls by the modified body of the called procedure. The resulting program will be faster and larger. Usually a limit is set on the size increase of the program. If this limit is set to zero, only calls that do not enlarge the program are substituted (i.e. calls to very small procedures or to procedures that are only called once). The goal is to expand those calls that maximize the speed increase of the program while obeying the size constraint. Scheifler [19] shows that this problem is NP-complete, but suggests a greedy algorithm that repeatedly chooses the call with the highest execution time gain relative to the size increase. Our main interests lie in finding a machine independent way to estimate the size and speed changes.

The increase in code size is determined by the size of the called procedure. The gain in execution time depends on the overhead introduced by the call and on how many times the call is executed dynamically.

If a call to a procedure P is expanded in line, the following instructions need not be executed:

(1) the pushing of the parameters on the stack;
(2) the subroutine jump ("jsr");
(3) the instructions in the procedure prolog of P; these instructions save registers, initialize the new frame pointer (local base) and allocate stack space for local variables;
(4) the procedure epilog instructions, which restore the register and frame pointer and release the space for local variables;
(5) the procedure return ("rts");
(6) the stack cleanup (removal of actual parameters).

Consequently, the savings in execution time are not the same for all procedures. Procedures having parameters and local variables will be more profitable. Inline Substitution may also gain indirectly, as it often introduces new opportunities for other optimizations. Especially calls with constants as parameters may lead to further optimizations.

The relative gain of a call is computed by looking at:

—the factors that determine the gain of one invocation (the presence of local variables, parameters, and constant parameters);
—whether the call appears inside a loop (such calls are expected to be executed more frequently);
—the number of EM instructions of the called procedure.

Summarizing, the only piece of information about the target machine that is really needed is the relation between the size of EM instructions and the number of bytes of the target machine. This figure can be obtained by compiling a set of sample programs.

7.10 *Register Allocation*

The Register Allocation technique tries to improve the normal register allocation scheme of ACK compilers. Besides local variables it also puts constants and addresses of variables and procedures in a register. Accessing a global variable indirectly through a register is usually more efficient than accessing it via its address. An even higher gain might be obtained by putting the variable itself in a register, but this approach is very complex and dangerous, as global variables are accessible from more than one procedure. The optimizer also puts different entities in the same register during different parts of the procedure, or during the same part provided that the entities are never live simultaneously.

The global optimizer knows (from the machine descriptor file) the number and types of the registers of the target machine. A register can be of the types floating point, pointer (address), or general. A fixed part of the register set is reserved for the code generator, to hold the frame pointer and to evaluate expressions. On the 68000 three data registers and four address registers (including

the stack pointer and frame pointer) are claimed by the code generator, so the global optimizer has four pointer (address) registers, five general (data) registers and no floating point registers.

For every entity the optimizer estimates the benefits and costs of putting the entity in a register. The following information can be obtained from the machine descriptor file:

—the costs of initializing a register with a value;
—the gain of using a register instead of an entity
—the costs of saving and restoring registers (at procedure entry and exit).

The costs of initializing a register with a certain value can be estimated reasonably accurate. The gain of using a register often depends on the context. Some average number is used for every pair (register type, kind of entity).

The save/restore costs are used to assure that the profits of using registers are not outweighed by the increase of the procedure call overhead. As several entities may reside in the same register, these costs should not be attributed to a specific entity. The optimizer first neglects these costs, but checks afterwards if the total profits of all entities assigned to the same register do pay off the overhead.

Register allocation is done separately for every procedure. The Register Allocator first makes a list of all entities (variables, constants, procedures) that are frequently used within the current procedure. For every such entity it must be decided during which parts of the procedure it should best be put in a register. Such a portion of a procedure is called a *timespan*. A timespan may, for example, consist of a loop or an entire procedure. The optimizer determines a set of candidate timespans for every entity, resulting in a list of (entity, timespan) pairs.

The profits of every pair are computed, which is the gain in time and space of putting the entity in a register during the timespan. The gain in code space is computed as the product of the number of occurrences in the timespan and the average space savings of one occurrence, as expressed in the machine descriptor file. It is inherently more difficult to estimate the gain in execution time, as it strongly depends on the dynamic behavior of the program. The program of Fig. 8 contains 4 calls to procedure p. Dynamically p may not be called at all. Putting the address of p in a register for the duration of the case-statement probably leads to an increase of execution time cause by the initialization of the register.

The optimizer uses a shortest path algorithm to determine the minimal number of dynamic occurrences and avoids doing register allocations that may be disadvantageous. In assessing the time savings, the loop depth of every occurrence is taken into account.

Equipped with all this information, the register allocator assigns registers to entities. Different algorithms are used for optimizing time and space.

The algorithm for space optimization is greedy. It repeatedly selects the (entity, timespan) pair with the highest profits among those pairs that can be assigned a register. Two pairs (X, S) and (Y, T) can be put in the same register if X and Y are never live simultaneously during the intersection of S and T. Even if all register are occupied, there still may be pairs that can be assigned a register.

The time optimization algorithm is based on the property of programs to spend most of their time in loops. First, only those (entity, timespan) pairs are considered of which the timespan consists of a loop. Pairs are selected in decreasing order of their gain. Second, all remaining pairs are considered.

8. MEASUREMENTS

This section contains some experimental results of the global optimizer. We have restricted ourselves to execution time optimizations. All experiments described here were carried out on a 68000.

The overall effectiveness of the global optimizer is difficult to estimate, as it strongly depends on its input program. Traditionally, optimizers are analyzed using a set of benchmark programs, typically small toy programs. We have found that measurements using such input are highly inaccurate, as it is often possible to dramatically speedup the program by just a single optimization.

```
case random of
  16891:  ... p(...); ...
  10965:  ... p(...); ...  p(...); ...
  26591:  ... p(...); ...
  else    ....;
end;
```

Fig. 8. Difference between time and space optimization.

For example, for bubble sort [6], just substituting the single call to the procedure "swap" (which exchanges two variables) cuts the execution time in half. Strength Reduction on a straightforward matrix multiplication program written in Pascal reduces execution time almost 80%. Artificially constructed benchmark programs, such as the dhrystone test [34], often contain far more cases for optimizations like Dead Code Elimination than normal programs. For more realistic programs, such easy gains are less likely to occur. For completeness sake, Table 1 contains the execution times and speedups for some popular benchmark programs. The dhrystone test was written in "C"; all remaining programs were written in Pascal.

An important issue is whether the programmer was anticipating the use of an optimizing compiler while developing his program. In our programming environment, the standard compiler performs no global optimizations, so most system programs are hand-optimized. As they are written in "C", there is ample opportunity for such optimizations by hand, for example by using macros instead of procedures, pointer variables instead of array indices, and by supplying accurate register declarations. Needless to say, a global optimizer will be less effective for such programs.

For these reasons, we decided to use the global optimizer itself as test input. The optimizer consists of fairly large programs (passes). Its codes uses all the efficient constructs of the "C" language, but it is less hand-optimized than the usual system utilities. All passes of the optimizer were compiled with the 68000 "C" compiler of the compiler kit, both with and without invoking the global optimizer. The resulting passes were used to optimize a fairly large (over 1700 lines) program. The execution times of the passes are shown in Table 2.

Taking the average of these two tests, we come to the conclusion that the global optimizer speeds up program execution by 16–39%, depending on the input.

Table 1. Results for some popular benchmark programs

Program	(1) Not opt. (sec)	(2) Optimized (sec)	(3) Ratio (2)/(1)
Bubble sort	2.9	1.2	0.41
Quick sort	2.6	2.4	0.92
Matrix mult.	13.8	2.6	0.19
8 Queens	9.0	4.9	0.54
Towers hanoi	7.4	6.6	0.89
Dhrystone	38.2	27.5	0.72
Total	73.9	45.2	0.61

Table 2. Results for passes of global optimizer

Pass	(1) Not opt. (sec)	(2) Optimized (sec)	(3) Ratio (2)/(1)
Intermediate code construction	6.3	4.7	0.75
Control flow analysis	10.1	7.8	0.77
Inline substitution	27.8	23.0	0.83
Control flow (repeated)	33.5	28.6	0.85
Common subexpressions	35.5	28.9	0.84
Strength reduction	24.3	20.5	0.84
Stack pollution	22.2	19.1	0.86
Cross jumping	31.7	28.1	0.88
Constant/variable folding	127.8	95.1	0.74
Dead code elimination	72.2	58.7	0.81
Register allocation	418.8	361.9	0.86
External EM code construction	12.6	10.6	0.84
Total	845.5	706.1	0.84

9. DISCUSSION

In the previous sections we looked at the problems associated with language- and machine-independent global optimization on intermediate code. We showed how these problems were dealt with in the EM global optimizer. The tendency was to use simple, heuristic solutions, rather than to aim at the highest possible degree of effectiveness. In this section we will review some aspects of our approach.

The Control Flow Graph representation of the program has proven to be a suitable one. Its main benefit is the fact that the flow of control is described using only one concept, the successor/predecessor relationship. The local code of the nodes of the graph (the basic blocks) was represented as a linear list, which is sometimes convenient, but sometimes not. For example, for Inline Substitution, the actual parameters of a procedure call must be obtained by splitting the linear code into several expressions. This essentially involves parsing the EM code, to detect the bounds of the expressions. If the local code would be represented as a tree, this would not be needed. On the other hand, techniques like Cross Jumping and Common Subexpression Elimination can be implemented conveniently using straight line code.

We agree with Goos and Waite [7, p. 327] that a high level representation such as a structure tree is not well suited for most optimizations. (Note that we are not discussing language specific optimizations, such as eliminating runtime checks [25, p. 23] or finding efficient implementations for Ada generic units [35]). The lack of detail is fatal for optimizations like Cross Jumping and Common Subexpression Elimination. A potential advantage of a tree is the possibility to avoid control flow analysis, as the flow of control is already implicit in the structure of the tree. However, in the presence of explicit goto's or exits from loops or procedures, much of this disadvantage disappears. Mintz *et al.* [36] perform data flow analysis on parse trees, but generate explicit flow graphs when goto's are encountered.

Optimizing assembly or object code seems to be attractive with respect to effectiveness, as the effects of a transformation can be determined with high accuracy. Yet, the price one has to pay to achieve a machine-independent optimizer will likewise be higher.

EM has a rather extensive instruction set, containing many special case instructions that are found in real computers too. For this reason, EM is sometimes called a "union machine" [11] as opposed to "intersection machines" that provide only the most essential operations. The special case instructions are especially useful for peephole optimizations, as shown in [3]. For global optimization it is convenient to have, for example, one instruction to access a local variable, rather than a sequence of basic operations that explicitly compute the address of the variable and the access it indirectly. On the other hand, special case instructions like INCREMENT and DECREMENT are of little use to a global optimizer and cause some minor inconveniences. It is also true that the instruction set will never be sufficient anyway. The absence of an exponentiation operator is well argumented by noting that almost all target machines and many source languages don't have such an operator either, yet it would be useful for optimizing the languages that do support the operator. As a whole, the EM instruction set is reasonably suited for optimization.

10. SUMMARY

This paper describes the design of a language- and machine-independent global optimizer. The optimizer is part of the Amsterdam Compiler Kit (ACK). The approach taken by ACK is to do global optimizations on a low level intermediate code (EM), so the optimizer is useful for a large class of programming languages and machines. ACK is compared with other compiler systems that use different kinds of intermediate code. The optimizations done by the ACK global optimizer are explained briefly. The problems encountered with optimizing EM code are discussed in detail. It is found that the effectiveness of an optimizer operating on an intermediate code depends on the level of detail of the intermediate code, and on the information about the source language, source program, and target machine that is available to the optimizer. It is shown that EM has sufficient detail and contains enough source program information. Some information about the target machine is made available to the optimizer via a machine descriptor file. The implementation of

the optimizer is described in some detail. Finally, some measurements are discussed to estimate the effectiveness of the optimizer.

Acknowledgement—This research was supported by the Stichting Technische Wetenschappen (STW) under grant VWI00.001.

REFERENCES

1. Lowry E. S. and Medlock C. W. Object code optimization. *Commun. ACM* **12**(1), 13–22 (January 1969).
2. Wulf W. A. PQCC: A machine-relative compiler technology. CMU-CS-80-144, Carnegie-Mellon University, Pittsburgh (25 September 1980).
3. Tanenbaum A. S., Staveren H. van and Stevenson J. W. Using peephole optimization on intermediate code. *ACM Trans. Progr. Lang. Sys.* **4**(1), 21–36 (January 1982).
4. Perkins D. R. and Sites R. L. Machine-independent Pascal code optimization. *SIGPLAN Not.* **14**(8), 201–207 (August 1979).
5. Auslander M. A. and Hopkins M. E. An overview of the PL.8 compiler. *SIGPLAN Not.* **17**(6), 22–31 (June 1982).
6. Chow F. C. A portable machine-independent global optimizer—Design and measurements. Computer Systems Laboratory, Stanford University (December 1983).
7. Waite W. M. and Goos G., *Compiler Construction*, Springer, New York (1984).
8. Brosgol B. M. TCOLAda and the middle end of the PQCC Ada compiler. *SIGPLAN Not.* **15**(1), 101–112 (November 1980).
9. Johnson S. C. A tour through the portable C compiler. In *Unix Programmer's Manual*. 7th edn. Bell Laboratories, Murray Hill, N.J. (January 1979).
10. Sethi R. and Ullman J. D. The generation of optimal code for arithmetic expressions. *J. ACM* **17**(4), pp. 715–728 (October 1970).
11. Davidson J. W. Simplifying code generation through peephole optimization. Ph.D. thesis, Department of Computer Science, University of Arizona (December 1981).
12. Davidson J. W. and Fraser C. W. Code selection through object code optimization. Department of Computer Science, University of Arizona (November 1981).
13. Davidson J. W. and Fraser C. W. Automatic generation of peephole optimizations. *SIGPLAN Not.* **19**(6), 111–116 (June 1984).
14. Leverett B. W., Cattell R. G. G., Hobbs S. O., Newcomer J. M., Reiner A. H., Schatz B. R. and Wulf W. A. An overview of the Production-Quality Compiler-Compiler Project. CMU-CS-79-105, Carnegie-Mellon University, Pittsburgh (1979).
15. Steel T. B. UNCOL: The myth and the fact. *A. Rev. Autom. Program.* **2**, 325–344 (1960).
16. Tanenbaum A. S., Staveren H. van, Keizer E. G. and Stevenson J. W. A practical toolkit for making portable compilers. *Commun. ACM* **26**(9), 654–660 (September 1983).
17. Tanenbaum A. S., Staveren H. van, Keizer E. G. and Stevenson J. W. A Unix toolkit for making portable compilers. *Proc. USENIX Conf.*, Toronto, Canada, Vol. 26, pp. 255–261 (July 1983).
18. Tanenbaum A. S., Staveren H. van, Keizer E. G. and Stevenson J. W. Description of a machine architecture for use with block structured languages, Rapport IR-81, Vrije Universiteit, Amsterdam (August 1983).
19. Scheifler R. W. An analysis of inline substitution for a structured programming language. *Commun. ACM* **20**(9), 647–654 (September 1977).
20. Kennedy K. A survey of data flow analysis techniques. In *Program Flow Analysis* (Edited by Muchnick S. S. and Jones D.). Prentice–Hall, Englewood Cliffs, N.J. (1981).
21. Allen F. E., Cocke J. and Kennedy K. Reduction of operator strength. In *Program Flow Analysis* (Edited by Muchnick S. S. and Jones D.). Prentice–Hall, Englewood Cliffs, N.J. (1981).
22. Wulf W. A., Johnsson R. K., Weinstock C. B., Hobbs S. O. and Geschke C. M. *The Design of an Optimizing Compiler*. Elsevier, New York (1975).
23. Aho A. V. and Ullman J. D. *Principles of Compiler Design*. Addison–Wesley, Reading, Mass. (1978).
24. Leverett B. W. Register allocation in optimizing compilers. Ph.D. thesis, CMU-CS-81-103, Carnegie-Mellon University, Pittsburgh (February 1981).
25. Kirchgaesner W., Uhl J., Winterstein G., Goos G., Dausmann M. and Drossopoulou S. An optimizing Ada compiler. Institut fur Informatik II, Universitat Karlsruhe (February 1983).
26. Morgan T. M. and Rowe L. A. Analyzing exotic instructions for a retargetable code generator. *SIGPLAN Not.* **17**(6), 197–204 (June 1982).
27. Ichbiah J. D. Rationale for the design of the Ada programming language. *SIGPLAN Not.* **14**(6) (June 1979).
28. Kernighan B. W. and McIlroy M. D. *Unix Programmer's Manual*, 7th edn. Bell Laboratories, Murray Hill, N.J. (January 1979).
29. Spillman T. C. Exposing side-effects in a PL/I optimizing compiler. In *Information Processing 1971*, pp. 376–381. North-Holland, Amsterdam (1971).
30. Kernighan B. W. and Ritchie D. M. *The C Programming Language*. Prentice–Hall, Englewood Cliffs, N.I. (1978).
31. Bal H. E. The design and implementation of the EM global optimizer. Rapport IR-99, Vrije Universiteit, Amsterdam (March 1985).
32. Prabhala B. and Sethi R. Efficient computation of expressions with common subexpressions. *J. ACM* **27**(1), 146–163 (January 1980).
33. Lengauer T. and Tarjan R. E. A fast algorithm for finding dominators in a flowgraph. *ACM Trans. Program. Lang. Syst.* **1**(1), 121–141 (July 1979).
34. Weicker R. P. Dhrystone: A synthetic systems programming benchmark. *Commun. ACM* **27**(10), 1013–1030. (October 1984).
35. Bray G. Implementation implications of Ada generics. *Ada Lett.* **III**(2), 62–71 (September 1983).
36. Mintz R. J., Fisher G. A. and Sharir M., The design of a global optimizer. *SIGPLAN Not.* **14**(9), 226–234 (September 1979).

About the Author—HENRI E. BAL was born in The Hague, The Netherlands. He received a Master's degree in Mathematics from Delft Technical University in 1982. Since 1982 he has been working at the Mathematical and Computer Science Department of the Vrije Universiteit of Amsterdam. His research interests include programming languages, compiler-building technology, and distributed computing systems. He participated in the Amsterdam Compiler Kit project. At present, he is doing research on languages and applications for distributed computing systems.

About the Author—ANDREW S. TANENBAUM was born in New York City. He received a Bachelor's degree from M.I.T., and a Ph.D. from the University of California at Berkeley. Since 1971 he has been on the faculty of the Vrije Universiteit in Amsterdam. Dr Tanenbaum's current research interests are compiler-building technology and distributed operating systems. He is the author of 3 books and more than 40 papers. He is principal architect of the Amsterdam Compiler Kit, which is now used at universities and companies on 5 continents. Dr Tanenbaum has worked for IBM and consulted for AT&T Bell Laboratories, and has lectured in more than a dozen countries around the world.